Methods for High Performance Graph Processing

Mikhail Chernoskutov

Krasovskii Institute of Mathematics and Mechanics,
Ural Federal University,
Yekaterinburg, Russia
mach@imm.uran.ru

Abstract. This paper describes two methods for accelerating the processing of a large graph distributed in the memory of multiple nodes. The first allows to substantially reduce overheads connected with data transfer between different nodes. The second is designed to reduce workload imbalance amongst computational threads. Both methods are integrated into a breadth-first search algorithm and more than triple its performance.

Keywords: data intensive applications, graph processing, parallel algorithms.

1 Introduction

Graph algorithms are used in various fields of science and engineering applications. In many cases, big graphs can be processed in parallel by computational systems with multiple nodes [1]. However, the parallelization efficiency is impaired by intensive memory access and unknown (in general) exact location of data on systems with distributed memory. These obstacles turn graph algorithms into typical data-intensive applications [2].

Intensive memory access pattern is a crucial bottleneck in implementations of parallel graph algorithms because (in many cases) there is only a small number of computational operations in such classes of algorithms. On the other hand, graph algorithms tend to have a lot of operations related to access to small pieces of data in different parts of memory. Thus, they are less demanding as far as CPU capabilities are concerned but require much in the way of efficiency and bandwidth of the data transfer bus.

The fact that the distribution of data is not known in advance dramatically complicates efficient implementation of multi-node parallel implementations of graph algorithms. In the general case, the program might have to search for the data on some particular vertex across all the computational nodes.

The object of this research is a parallel breadth-first search algorithm on graphs with small diameter (generally, no more than 10) and skewed degree distribution. Such graphs arise in, for instance, social networks analysis, various mathematical and physical simulation applications [3], etc. The main feature of

these graphs is a relatively small number of vertices with highest degrees and large number of vertices with small degrees (with only few incident vertices).

2 Parallel Breadth-First Search

Breadth-first search may be parallelized with the aid of level-synchronous algorithms. These algorithms process each level (or iteration) separately and independently from each other. It means that (in case of breadth-first search) processing of the level $N + 1$ begins after the processing of the level N has been finished, while each of these levels (i.e., all vertices and edges of the same level) may be processed in parallel.

Presently, there are two most common types of level-synchronous breadth-first search algorithms:

– direct traversal (top-down approach);
– inverse traversal (bottom-up approach [4]).

The direct graph traversal is the standard version of breadth-first search algorithm of this kind. It assumes that the vertices that are active on the current iteration would mark all their neighbors. The pseudocode of a parallel version of breadth-first search with the top-down traversal direction is presented on fig. 1.

In lines 1–4, the array of distances to source vertices is initialized. The initialization is followed by the main loop, which repeats while there exist unmarked vertices in the graph. First, in lines 6–14, the vertices stored at the current node are marked. Then, in lines 16 and 17, messages to other nodes are sent and received (point-to-point); these messages contain the data on vertices that must be marked on those nodes. Finally, in lines 19–22, the vertices the data on which was received from other nodes are marked. At the end of each iteration, the level counter in line 23 is incremented by one and then, at line 24, the algorithm checks if there are still unmarked vertices.

The bottom-up implementation of breadth-first search algorithm assumes that inactive vertices will be looking for active vertices amongst their neighbors. In case of presence of such type of vertices on the current iteration, the vertex is to mark itself. The pseudocode of parallel breadth-first search with inverse traversal is presented on fig. 2.

Like in the previous pseudocode, in lines 1–4, the breadth-first search is initialized. In lines 18 and 19, the level number is updated and it is tested if the algorithm is to terminate. However, there is a big difference in the vertex marking procedure—in case of bottom-up traversal, it is crucial to know the current state of all active vertices in the graph. It is most convenient to do this by means of a bitmap the length of which equals the number of vertices in the graph. Therefore, at each iteration, the data is synchronized by means of updating the bitmap through collective MPI communications (line 15 and 16).

```
1  for each u in dist
2    dist[u] := −1
3  dist[s] := 0
4  level := 0
5  do
6    parallel for each vert in V.this_node
7      if dist[vert] = level
8        for each neighb in vert.neighbors
9          if neighb in V.this_node
10           if dist[neighb] = −1
11             dist[neighb] := level + 1
12             pred[neighb] := vert
13           else
14             vert_batch_to_send.push(neighb)
15
16   send(vert_batch_to_send)
17   receive(vert_batch_to_receive)
18
19   parallel for each vert in vert_batch_to_receive
20     if dist[vert] = −1
21       dist[vert] := level + 1
22       pred[vert] := vert.pred
23   level++
24 while(!check_end())
```

Fig. 1. Parallel top-down breadth-first search algorithm pseudocode

```
1  for each u in dist
2    dist[u] := −1
3  dist[s] := 0
4  level := 0
5  do
6    parallel for each vert in V.this_node
7      if dist[vert] = −1
8        for each neighb in vert.neighbors
9          if bitmap_current.neighb = 1
10           dist[vert] := level + 1
11           pred[vert] := neighb
12           bitmap_next.vert := 1
13           break
14
15   all_gather(bitmap_next)
16   swap(bitmap_current, bitmap_next)
17
18   level++
19 while(!check_end())
```

Fig. 2. Parallel bottom-up breadth-first search algorithm pseudocode

3 Performance Engineering of Parallel Breadth-First Search

To increase the performance of parallel breadth-first search algorithms, one could suggest the following two methods:

– hybrid graph traversal;
– workload distribution across computational threads.

3.1 Hybrid Traversal

This method is a combination of different types of parallel breadth-first search algorithm for executing different iterations. In particular, the top-down approach characterized by large computational workload and data transfer overheads on iterations in the middle. At the same time, the first and last iterations in the top-down approach execute faster and have almost no data transfers. The situation is different with the bottom-up approach due to the use of collective MPI data transfer operations. In its case, data transfer overheads are almost the same on each iteration. However, marking the vertices on the first iterations takes much more time than on the iterations in the middle or later.

Taking into account the fact that the data produced on each iteration is the same (regardless of traversal direction), we suggest to combine different types of graph traversal (with smallest possible data transfer overheads on each iteration) to achieve maximal performance. In this study, we propose the following scheme:

– top-down on first two iterations;
– bottom-up on next three iterations;
– top-down on all other iterations.

3.2 Workload Distribution

In processing of graphs with skewed degree distribution, it is not known in advance (in the general case) which vertices will be processed by a particular computational thread, the only thing that may be known in advance is, for instance, the total number of vertices that have to be processed by the current thread. However, the total workload is determined not by the number of active vertices but by the number of edges incident to them. This leads to workload imbalance amongst threads and big overheads during the runtime of parallel level-synchronous algorithm.

To avoid workload imbalance, we suggest a transition from “looking” through a vertex array to “looking‘‘ through an array of edges. For this purpose, we logically divide the edges array into equal pieces holding *max_edges* elements. Each thread determines the corresponding vertex for all edges in every part of the edges’ array by using the *part_column* array, which contains the numbers of vertices incident to the first edge in the corresponding part of the edges’ array. The pseudocode for parallel filling of the *part_column* array is presented on fig. 3.

```
1  parallel for each i in V.this_node
2    first := V.this_node[i]
3    last := V.this_node[i+1]
4    index := round_up(first/max_edges)
5    current := index*max_edges
6    while(current < last)
7      part_column[index] := i
8      current := current + max_edges
9      index++
```

Fig. 3. Parallel filling of *part_column* array pseudocode

Pseudocode of a new version of breadth-first search algorithm that uses the *part_column* array is presented on fig. 4 (the top-down approach) and fig. 5 (the bottom-up approach).

4 Benchmarking

Both of the aforementioned methods were incorporated into a custom implementation of the Graph500 benchmark [5]. The kernel of this benchmark represents parallel breadth-first search on a big graph distributed in the memory of multiple nodes. The size of the graph is determined by the scale parameter, which is the logarithm base 2 of the number of vertices in the graph. The average degree of all vertices is 16. The main performance metric of this benchmark is the number of edges traversed per second (TEPS).

We developed a custom implementation that uses MPI (one process per computational node) for sending and receiving messages across all nodes and OpenMP (eight threads per node) to deal with shared memory within each node.

Benchmarking was carried out for graphs with different sizes on 1-, 2-, 4-, and 8-node configurations of the Uran supercomputer (located at the Krasovskii Institute of Mathematics and Mechanics). Each node had Intel Xeon X5675 CPU and 46 GB DRAM. Performance of the custom implementation was compared with the following reference implementations provided by Graph500:

- simple (represents top-down breadth-first search);
- replicated (represents bottom-up breadth-first search).

Benchmarking results are presented on fig. 6. As seen on the figure, our custom implementation substantially outperforms the simple and replicated implementations. In addition, in case of 8 nodes, it is clearly seen that that the custom implementation scales much better than its counterparts (scalability of all implementations presented on fig 7).

```
1  // preparation...
2  parallel for each i in part_column
3    first_edge := i*max_edges
4    last_edge := (i+1)*max_edges
5    curr_vert := part_column[i]
6    for each edge in [first_edge;last_edge)
7      if neighbors of curr_vert in [first_edge;last_edge)
8        if dist[curr_vert] = level
9          for each k in neighbors of curr_vert
10           if dist[k] = -1
11             dist[k] := level + 1
12             pred[k] := curr_vert
13     curr_vert++
14 // data synchronization...
```

Fig. 4. Parallel top-down breadth-first search algorithm pseudocode (with workload balancing)

```
1  // preparation...
2  parallel for i in part_column
3    first_edge := i*max_edges
4    last_edge := (i+1)*max_edges
5    curr_vert := part_column[i]
6    for each edge in [first_edge;last_edge)
7      if neighbors of curr_vert in [first_edge;last_edge)
8        if dist[curr_vert] = -1
9          for each k in neighbors of curr_vert
10           if bitmap_current.k = 1
11             dist[curr_vert] := level + 1
12             pred[curr_vert] := k
13             bitmap_next.vert := 1
14             break
15     curr_vert++
16 // data synchronization...
```

Fig. 5. Parallel bottom-up breadth-first search algorithm pseudocode (with workload balancing)

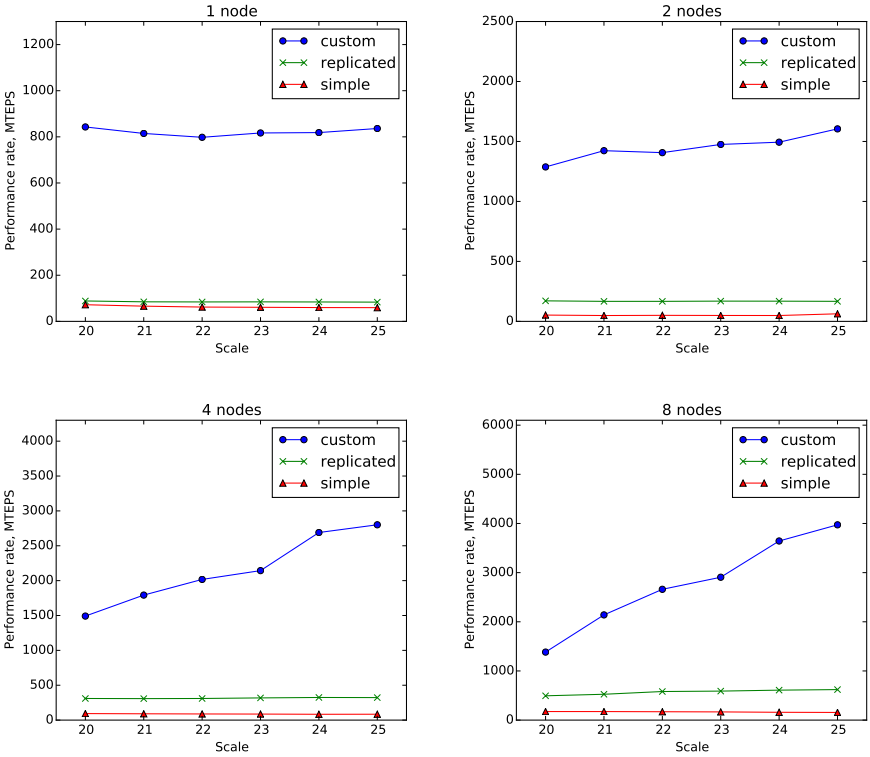

Fig. 6. Performance comparison

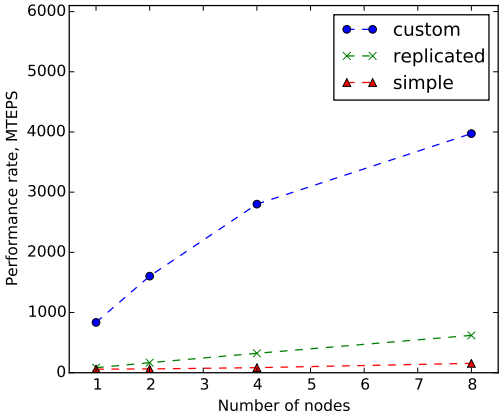

Fig. 7. Scalability comparison

5 Conclusion

Attempts at efficient parallelization of the breadth-first search algorithm with skewed degree distribution are hampered by the workload imbalance amongst computational threads and large amounts of data transfer only on few select iterations of the algorithm. This forms a bottleneck that makes it much more hard to make a high-performance implementation for this algorithm.

In this paper, we suggest a couple of methods for workload balancing and traversal hybridization, which allow to increase the performance (it is more than three times higher) of the parallel level-synchronous breadth-first search in comparison with the reference top-down and bottom-up procedures.

In our future work, we intend to focus on the research in scalability the suggested algorithm and testing it on graphs obtained from real-world applications. Another important task is to modify our custom implementation to use computational accelerators to improve its performance.

Acknowledgments. The reported study was partially supported by RFBR, research project No. 14-07-00435. The experiment was performed on the Uran supercomputer.

References

1. Mark E.J. Newman. The structure and function of complex networks. SIAM review, 45(2):167256, 2003.
2. Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. Parallel Processing Letters, 17(01):520, 2007.
3. Bruce Hendrickson and Jonathan W. Berry. Graph analysis with high-performance computing. Computing in Science and Engg., 10(2):1419, March 2008.
4. Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 12, pages 12:112:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
5. Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the graph 500. 2010.